Available online at www.sciencedirect.com
SCIENCE DIRECT
Reliability Engineering and System Safety 89 (2005) 17–32
RELIABILITY ENGINEERING & SYSTEM SAFETY
www.elsevier.com/locate/ress

A unified tool for performance modelling and prediction

Stephen Gilmore*, Leïla Kloul[1]

Laboratory for Foundations of Computer Science, The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, Scotland EH9 3JZ, UK

Available online 5 January 2005

Abstract

We describe a novel performability modelling approach, which facilitates the efficient solution of performance models extracted from high-level descriptions of systems. The notation which we use for our high-level designs is the Unified Modelling Language (UML) graphical modelling language. The technology which provides the efficient representation capability for the underlying performance model is the multi-terminal binary decision diagram (MTBDD)-based PRISM probabilistic model checker. The UML models are compiled through an intermediate language, the stochastic process algebra PEPA, before translation into MTBDDs for solution. We illustrate our approach on a real-world analysis problem from the domain of mobile telephony.

Keywords: Performance modelling; Multi-terminal binary decision diagram; Performance evaluation process algebra

1. Introduction

Distributed, mobile and global computing environments provide robust development challenges to practising software system developers. Working with rapidly changing implementation technology means that developers often must spend some of their development time finding and correcting errors in the software libraries and APIs which they use. Fortifying this difficulty is the arduous terrain of dynamic distributed systems where the difficulty of replaying a communication sequence which led to a system fault confounds the process of detecting and correcting implementation errors.

In this setting, application developers rarely wish to expend the investment of time which would be needed to build and analyse a performance model of the system which they are developing. The concepts and the modelling languages of performance analysis are relatively unfamiliar to software developers and when already faced with a generous range of other difficulties in the development process, early predictive performance analysis can easily be overlooked. However, this is an imprudent practice. If performance design flaws are found early in the development process then they can be corrected at a relatively low cost. In contrast, if they are found after the development process is long underway then they may be very expensive or even unrealistic to repair.

Performance impacts on dependability because application technology with poor or unpredictable performance cannot be depended on to perform when it is most needed. Thus, there is a continuum between performance analysis and dependability analysis encompassing the federated study of these two subjects known as *performability analysis* [1]. We discuss performability analysis here and show how custom performability requirements can be checked against high-level application models.

Our belief is that a performability-aware methodology for the effective development of global, mobile or high-end distributed systems should provide at least the following two features: a convenient high-level modelling notation for expressing performance models; and efficient solution methods for realistic models of complex systems. Unfortunately, these two requirements are

* Corresponding author. Tel.: +44 131 6505 189; fax: +44 131 667 7209.
E-mail address: stg@inf.ed.ac.uk (S. Gilmore).
[1] On leave from PRiSM, Université de Versailles, 45, av. des Etats-Unis, 78000 Versailles, France.

0951-8320/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ress.2004.08.004

often at variance. In order to access state-of-the-art solution methods one usually additionally needs to master sophisticated representation and analysis methods which sometimes are inconvenient or troublesome to use. Conversely, high-level modelling platforms devote much of their efforts to providing reliable graphical editors and supporting document and IDE infrastructure and this can come at the expense of equipping them with complementary analysis tools.

We provide a structured performance-engineering platform for this problem domain by connecting a *specification environment* (SENV) and a *verification environment* (VENV) so that each may communicate with the other. The SENV and VENV are connected by a bridge, which consists of two categories of software tools. These are:

- *extractors* which translate designs from the SENV into inputs for the VENV, omitting any aspects of the design which are not relevant for the verification task at hand;
- *reflectors* which convert the results from the analysis performed by the VENV back into a form which can be processed and displayed by the SENV.

A series of extractors can be chained together to provide a path from one specification formalism to another. Similarly, reflectors can be chained together in order that the results of one analysis process may be presented back in the format of another. A process of *extractor/reflector chaining* is used here to connect a specification environment to multiple verification environments. We use the ArgoUML design environment [2] as our SENV and the performance evaluation process algebra (PEPA) Workbench [3] and the PRISM probabilistic symbolic model checker [4] both play the role of VENVs. ArgoUML provides the Unified Modelling Language (UML) [5] as its modelling language. The PEPA Workbench and PRISM both support the PEPA stochastic process algebra [6]. PRISM additionally supports a state-based language based on the Reactive Modules formalism of Alur and Henzinger [7].

In Ref. [8], we have presented the algorithm for deriving a PEPA model from a UML model using the PEPA Workbench as the unique verification environment. In this paper, we use both the PEPA Workbench and the PRISM probabilistic symbolic model checker. Moreover, we show how to apply this new approach to a realistic example, which is the hierarchical wireless network.

Structure of this paper. In Section 2, we describe some of the background to this work, providing a summary of UML, PEPA and PRISM modelling. In Section 3, we describe the software architecture of the system, as an integrated set of components. We give details of the implementation, providing an explanation of how these components work together. In Section 4, we discuss the correctness of our translation from UML to PEPA and PRISM calling attention to cases where some care has been needed. In Section 5, we present our case study, showing the approach applied to a realistic example. In Section 6, we survey related work. Conclusions are presented in Section 7.

2. Background

2.1. Unified modelling language

The UML is an effective diagrammatic notation used to capture high-level designs of systems, especially object-oriented software systems. A UML *model* is represented by a collection of diagrams describing parts of the system from different points of view; there are seven main diagram types. For example, there will typically be a *static structure diagram* (or *class diagram*) describing the classes and interfaces in the system and their static relationships (inheritance, dependency, etc.). State diagrams, a variant of Harel state charts, can be used to record dynamic behaviour. Interaction diagrams, such as sequence diagrams, are used to illustrate the way objects of different classes interact in a particular scenario. As usual we expect that the UML modeller will make a number of diagrams of different kinds. Our analysis is based on state and collaboration diagrams.

We have introduced performance information in the state diagrams such that each transition in these diagrams is labelled with a pair a/rate(r) where a is the action type executed and r is an exponentially distributed rate associated with this action. We often simplify the representation of the transition labels in order to save space writing this as a/r.

As an example of a UML model consider a client–server system where a client sends requests to the server to undertake computationally expensive tasks. Once the client has sent a request, it should wait for the server to reply before continuing with its own work. We denote by λ the occurrence rate of client requests and by μ the service rate at the server. The UML model of such a system is composed of a class diagram, two state diagrams and a collaboration diagram.

The class diagrams introduce the components of the system. With each class is associated a state diagram which describes the state-to-state transition behaviour of each instance of the class.

- The first state diagram (Fig. 1) describes the states of the server. We assume that the server may be either *idle* or *busy* and the transition from the first state to the second one corresponds to the arrival of a request from the client. Note that the rate associated with the action

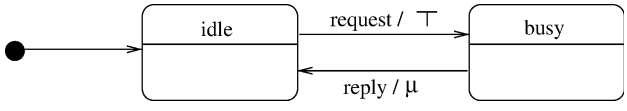

Fig. 1. The state diagram of *Server*.

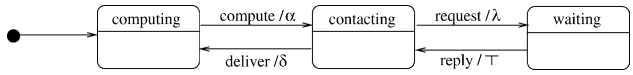

Fig. 2. The state diagram of *Client*.

request at the server level is unspecified (⊤) since the requests are generated by the client. The transition to the *idle* state corresponds to the completion of the request processing and the reply to the client, represented formally as action *reply* with rate μ.

- The second state diagram (Fig. 2) models the behaviour of a client. We consider three possible states *Computing*, *Contacting* and *Waiting*. In the first state, the client is engaged in local computation, action *compute* with rate α. When greater compute power is needed, he contacts the server (state *Contacting*) and issues the request, action *request*. After a certain waiting time, he receives a response from the server with action *reply*. The rate at which replies are produced is not specified by the client but by the server. From the client's point of view, this rate is unspecified (⊤).
- The collaboration diagram in Fig. 3 depicts the interactions between the system components, *Server* and *Client*. We assume that we have two clients in the system, which means two instances of *Client*. The diagram shows that both clients must synchronize with the server, but it does not make explicit the moment or the actions on which they must synchronize. There is no association between *client*1 and *client*2, meaning that they behave independently from each other and have no common synchronisation points.

  In general, a UML collaboration diagram describes a typical operational configuration of the system, specifying the connections between the instances of the classes of the system. The UML class diagram has described the classes in the system, but not the number of instances of these which are active, nor the connections between them. The number of instances of each component is crucial information for the performance modelling process; a system with two servers is not the same as one with twenty. The collaboration diagram contains this information in the form of a directed graph linking together instances of the classes in the system.

2.2. Performance evaluation process algebra

In PEPA [6], a system is viewed as a set of *components* which carry out *activities* either individually or in co-operation with other components. Activities, which are private to the component in which they occur, are represented by the distinguished action type, τ. Each activity is characterized by an *action type* and a duration, which is exponentially distributed. This is written as a pair such as (α, r) where α is the action type and r is the *activity rate*. This parameter may be any positive real number, or may be unspecified. We use the distinguished symbol $\top$ to indicate that the rate is not specified by this component. This component is said to be *passive* with respect to this action type and the rate of the shared activity is defined by another component.

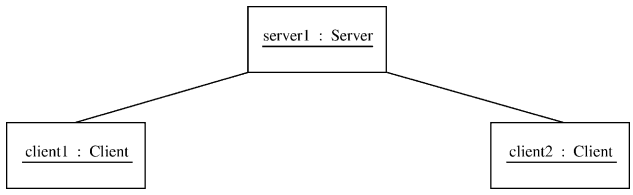

Fig. 3. The collaboration diagram.

PEPA provides a set of combinators, which allow expressions to be built which define the behaviour of components via the activities that they engage in. These combinators are presented in what follows.

Prefix $(\alpha, r)\cdot P$: Prefix is the basic mechanism by which the behaviours of components are constructed. This combinator implies that after the component has carried out activity (α, r), it behaves as component P.

Choice $P_1 + P_2$: This combinator represents a competition between components. The system may behave either as component P_1 or as P_2. All current activities of the two components are enabled. The first activity to complete distinguishes one of these components and the other is then discarded.

Co-operation $P_1 \bowtie_L P_2$: This describes the synchronization of components P_1 and P_2 over the activities in the co-operation set L. The components may proceed independently with activities whose types do not belong to this set. A particular case of the co-operation is when $L = \varnothing$. In this case, components proceed with all activities independently. The notation $P_1 \| P_2$ is used as a shorthand for $P_1 \bowtie_L P_2$ when L is empty. In a co-operation, the rate of a shared activity is defined as the rate of the slowest component.

Hiding P/L: This component behaves like P except that any activities of types within the set L are *hidden*, i.e. such an activity exhibits the unknown type τ and the activity can be regarded as an internal delay by the component. Such an activity cannot be carried out in co-operation with any other component: the original action type of a hidden activity is no longer externally accessible, to an observer or to another component; the duration is unaffected.

Constant $A \stackrel{\text{def}}{=} P$: Constants are components whose meaning is given by a defining equation: $A \stackrel{\text{def}}{=} P$ gives the constant A the behaviour of the component P. This is how we assign names to components (behaviours). An explicit recursion operator is not provided but components of infinite behaviour may be readily described using sets of mutually recursive defining equations.

For example, consider the client–server system described above. To model this system using PEPA, we need two components. Let *Client* and *Server* be the name of these

components. The PEPA model is the following.

$$Client_{cpg} \stackrel{def}{=} (compute, \alpha) \cdot Client_{ctg}$$

$$Client_{ctg} \stackrel{def}{=} (request, \lambda) \cdot Client_{wtg} + (deliver, \delta) \cdot Client_{cpg}$$

$$Client_{wtg} \stackrel{def}{=} (reply, \top) \cdot Client_{ctg}$$

$$Server_{idle} \stackrel{def}{=} (request, \top) \cdot Server_{busy}$$

$$Server_{busy} \stackrel{def}{=} (reply, \mu) \cdot Server_{idle}$$

If we assume that we have two clients in the system, the complete behaviour of the system is then obtained from the following equation:

$$System \stackrel{def}{=} (Client_{cpg} || Client_{cpg}) \underset{\mathcal{K}}{\bowtie} Server_{idle}$$

where $\mathcal{K} = \{request, reply\}$ is the set of activities on which *Client* and *Server* must synchronize.

In this work, such 'system equations' are automatically generated from the collaboration diagram in an input UML model. We treat collaboration diagrams as though they were undirected graphs because in the PEPA process algebra terms such as $Server \bowtie_L Client$ and $Client \bowtie_L Server$ are equivalent. We have previously presented the algorithm for deriving a PEPA system equation from such a collaboration diagram [8].

The transition system underlying the PEPA model gives the continuous-time Markov process represented by the model. The generation of this process is based on the derivation graph of the model in which syntactic terms form the nodes, and arcs represent the possible transitions between them. This derivation graph describes the possible behaviour of any PEPA component and provides a useful way to reason about a model. The use of the derivation graph is analogous to the use of the reachability graph in stochastic extensions of Petri nets such as GSPN [6]. The semantics of PEPA, presented in the structured operational semantics style, are given in Appendix.

2.3. PRISM

Specifications are constructed in PRISM by defining a collection of reactive modules which synchronise on shared activities. The state of each module is determined by a set of local variables. The models which we work with here are all obtained by compiling an input PEPA model. All of these have associated constants, which enumerate the states of the module and use a single local variable to record the current state. We developed the PEPA-to-PRISM compiler to allow us to analyse our PEPA models with PRISM. Our compiler has been incorporated into the latest release of PRISM (version 1.3) [9].

The behaviour of a reactive module is encoded by a list of guarded transitions, which name the activity performed and specify assignments to the local variables, which are to be carried out if the activity is performed. The PRISM notation for assignment is $x' = e$ meaning that the value of the variable x should be updated to hold the result of evaluating expression e. The PRISM specification of the server from our running example is shown below.

```
//Output from the PEPA-to-PRISM compiler
//Version 0.03.2 'Jean Armour Avenue'
//Model file: clientServer
stochastic
//The Server as a PRISM reactive modules
specification
const Server_busy=0;
const Server_idle=1;
module Server
  Server_STATE:      [0..1]      init
  Server_idle;
  [request]    (Server_STATE=Server_
  idle)→1:          (Server_STATE'=
  Server_busy);
  [reply]      (Server_STATE=Server_
  busy)→mu:         (Server_STATE'=
  Server_idle);
endmodule
```

The PRISM model checker accepts descriptions of discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markov decision processes (MDPs). However, all PEPA models define CTMCs so we mark this as a stochastic model using the PRISM keyword `stochastic`.

The definition of the Client is similar. The encoding of a choice in the PEPA model is written as a list of alternatives in the PRISM notation.

```
//Descriptive names for the local states
of
//this module, taken from the PEPA input
model
const Client_cpg=0;
const Client_ctg=1;
const Client_wtg=2;
module Client1
  Client1_STATE:     [0..2]     init
  Client_cpg;
  [compute]      (Client1_STATE=Cli-
  ent_cpg)→alpha:  (Client1_STATE'=
  Client_ctg);
  [deliver]      (Client1_STATE=Cli-
  ent_ctg)→delta:  (Client1_STATE'=
  Client_cpg);
  [request]      (Client1_STATE=Cli-
  ent_ctg)→lambda: (Client1_STATE'=
  Client_wtg);
  [reply]        (Client1_STATE=Cli-
  ent_wtg)→1:      (Client1_STATE'=
  Client_ctg);
endmodule
```

Modules may be duplicated and renamed. To make the second client in our example we would instantiate and rename the first client, renaming its local state variable to ensure that this is unique.

```
//We make another copy of the Client1
module
module Client2=Client1 [Client1_STA-
TE=Client2_STATE]
endmodule
```

Finally, the clients and server are composed together using synchronisation sets to specialise their activities, as in PEPA.

```
//The system equation
system
  ((Client_1 ||| Client_2) |[request,
  reply]| Server)
endsystem
//End of output from the PEPA-to-PRISM
compiler
```

We use PRISM to solve PEPA models for their stationary probability distribution. The first step of this process is generating the full state space of the system. This is compactly stored by PRISM as a multi-terminal binary decision diagram (MTBDD). The CUDD package [10] is used as a library, providing MTBDD data structures and algorithms to PRISM.

2.4. Analysis and model-checking

The PRISM model checker supports the analysis of probabilistic and stochastic systems by allowing a modeller to check a logical property against a model. The types of properties which we wish to check are performability questions assessing the likelihood of reaching a state where the system is either no longer available at all or—even if technically still available—its performance has been compromised to the point where the appreciated quality of service has fallen below the minimum standard demanded by the users of the system.

Several logics and several types of model are supported by the PRISM model checker. The appropriate logic for continuous-time Markov chains is Continuous Stochastic Logic (CSL) [11]. We have used PRISM here to perform CSL-based model-checking of our high-level models.

The syntax of CSL is:

$$\phi ::= true | false | a | \phi \wedge \phi | \phi \vee \phi | \neg \phi | \mathcal{P}_{\bowtie p}[\psi] | \mathcal{S}_{\bowtie p}[\phi]$$

$$\psi ::= \mathrm{X}\phi | \phi \mathrm{U}^{I} \phi | \phi \mathrm{U} \phi$$

where a is an atomic proposition, $\bowtie \in \{<, \leq, >, \geq\}$is a relational parameter, $p \in [0, 1]$ is a probability, and I is an interval of $\mathbb{R}$.

Paths of interest through the states of the model are characterised by the *path formulae* specified by $\mathcal{P}$. Path formulae either refer to the next state (using the X operator), or record that one proposition is always satisfied until another is achieved (the until-formulae use the U-operator). Performance information is encoded into the CSL formulae via the time-bounded until operator (U^{I}) and the steady-state operator, $\mathcal{S}$. By expressing properties of interest using path formulae we can check *interval-of-time* performability measures over our system. By expressing properties of interest with the steady-state operator we can determine *long-run* measures over the system.

In Section 3, we describe the process of connecting the ArgoUML, PEPA and PRISM tools.

3. The software architecture

Ours is a component-based software architecture in which we link substantial software tools with lightweight connectors called extractors and reflectors. This promotes significant code re-use and allows for clean interfaces between systems using formal description languages such as PEPA and PRISMs, reactive modules. The global software architecture in Fig. 4 shows the chaining process of the extractors and reflectors we use.

A performance modeller using our tool will design a system using a UML modelling environment such as ArgoUML or a similar UML tool. UML software tools produce XML-based model interchange files called XMI files.

The PEPA Workbench contains as components the PEPA Extractor and the PEPA Reflector. The former is first used to extract a PEPA model from the `.xmi` file containing the UML model. The resulting file is a `.pepa` file which is then compiled using the PEPA-to-PRISM compiler. The compiler provides PRISM with a log file in which PEPA local state identifiers have been mapped onto the numeric constants used in the reactive modules notation.

PRISM solves the PEPA model and produces a steady state probability vector. This is then reflected by the PRISM reflector as a PEPA steady state probability vector. To do so, the output from the PRISM tool onto the standard output

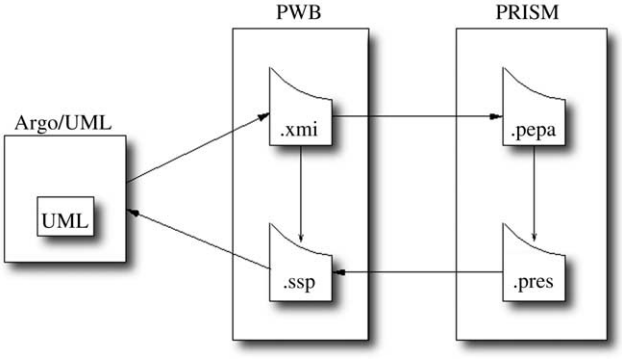

Fig. 4. Software architecture of the tool.

stream is captured and saved in a `.pres` (PRISM results) file. The PRISM Extractor reads the `.log` file and the `.pres` file and writes an `.ssp` file in the format used by the PEPA Workbench. The `.ssp` file contains *steady-state probabilities* indicating the probability of finding the system in one of its reachable states in the long run, after the initial transient effects of the initial system configuration have subsided.

Using some information in the `.xmi` file produced initially by the UML tool, the PEPA Workbench uses the PEPA Reflector to merge the results with the original input UML model to produce a modified `.xmi` file which includes the steady-state probabilities results of the performance analysis. This file can be loaded into the ArgoUML tool to present the results back to the UML performance modeller.

Our extractor and reflector software tools are implemented in the programming languages Java and Standard ML [12]. The Java components handle the processing of the zipped archives (in `.zargo` format) of UML models produced by the ArgoUML tool and the parsing of the XMI documents containing the UML models (in `.xmi` format). The DOM parser from the Java 1.4 `javax.xml` package is used to parse the XMI files. Much of the more complex processing is implemented in the Standard ML language. Standard ML is well-suited to symbolic processing problems and allows us to have greater confidence in the correctness of our manipulation of the UML, process algebra and reactive modules models than would have been possible otherwise. The Standard ML components interoperate with the Java ones because they are compiled with the ML-to-Java-bytecode compiler, MLj [13]. This means that they are seen from Java as a zipped archive of Java bytecode class files. Many other functional programming language compilers also target the Java platform in this way.

4. Correctness of the translation

Our analysis of a timed UML model proceeds by translating the given model into a PRISM reactive modules specification, obtained via a mapping into the PEPA stochastic process algebra. If either of these mappings is erroneous then all of the computational expense incurred in solving the model and all of the intellectual effort invested in the analysis and interpretation of the results obtained would at best be wasted. In general, interpreting a model with errors could lead to incorrectly concluding that the proposed system will perform with acceptable levels of availability and efficiency whereas this is in fact an erroneous conclusion deduced from erroneous results. For this reason, it is important to consider closely the correctness and validity of our translation process.

One part of the translation is more amenable to this form of analysis than the other, and we discuss this briefly here. It is possible to make concrete formal statements about the correctness of the translation from the intermediate PEPA model to the final PRISM model. Despite differences in form and expression, when fired against the operation semantics of their relative languages the PEPA and PRISM models generate isomorphic continuous-time Markov chains with equivalent states and transitions. This is the sense in which the equivalence of the two models can be formally stated.

However, this is only one part of the translation and the correctness of the whole depends also on the correctness of the initial translation from the UML model into PEPA. The same argument cannot be applied to this part of the translation because there is no canonical semantics for timed UML models with exponentially distributed delays which generates a CTMC. This prompts the question "In what sense does the intermediate PEPA model represent the input UML model?" and this question is more difficult to answer because there is on universally agreed formal semantics for UML models.

To this end, we have ensured that the generated PEPA model makes the least demands on the semantic interpretation of the UML diagram. That is, whatever semantics is used to interpret the diagrams it needs only to respect the state machine properties that each state machine is in one of its local states at any time, and that changes of states cannot be achieved by a means other than executing transitions. These properties are sufficient to ensure that there are no possible observations of the PEPA model which would not be possible observations of the UML state machine collaboration. This agreement can be reached in our work because we are working within a well-defined subset of the UML state diagrams language, which rules out syntactically those language features over which there is most debate as to their interpretation.

4.1. Special cases in the translation

In this section, we describe some of the places where special care needs to be taken to ensure that the PEPA input and PRISM output from the PEPA-to-PRISM compiler are in agreement.

The language accepted by the PEPA-to-PRISM compiler is a subset of the PEPA stochastic process algebra. The restrictions applied to the language are firstly that component identifiers can only be bound to sequential components (formed using prefix and choice and references to other sequential components only). Secondly, each local state of a sequential component must be named. For example, we would rewrite $P \stackrel{def}{=} (a, r) \cdot (b, s) \cdot P$ as

$$P \stackrel{def}{=} (a, r) \cdot P_2$$

$$P_2 \stackrel{def}{=} (b, s) \cdot P$$

These restrictions are relatively mild and are already enforced by the UML tools, which we use in any case. The diagrams which we can draw with our UML tool do not allow anonymous states such as $(b, s) \cdot P$ and they do not support the concurrency extension to state machines.

With regard to performance models one final restriction is applied: active/active synchronisations are not allowed. The reason for this is that the PRISM interpretation of these differs from the PEPA interpretation. In consequence, the results computed by PRISM from such a model would differ from the results computed by the PEPA Workbench.

PEPA synchronisations processed by the PEPA-to-PRISM compiler may have more than one passive participant. We use this fact crucially in encoding our performance measures as *witness components* in the model. These act as observers of the behaviour of the model, changing state to record significant events such as failures and repairs of servers, and giving names to important states such as the state where all of the servers have failed.

The restriction in force on synchronisations in the compiler is that each must have exactly one active component. This restriction is checked statically, without the need to explore the entire state space of the model. Because this is done statically, the PEPA-to-PRISM compiler will conservatively over-estimate the models in which an active/active synchronisation appears to occur. For example, the following model would be faulted by the compiler.

$$P \stackrel{def}{=} (a, r_1) \cdot P + (b, r_2) \cdot P_2 \quad Q \stackrel{def}{=} (c, r_4) \cdot Q$$

$$P_2 \stackrel{def}{=} (c, r_3) \cdot P$$

$$System \stackrel{def}{=} P \underset{\{b,c\}}{\bowtie} Q$$

The compiler infers the alphabets for each model component from its set of recursive definitions. It then classifies the activities in these alphabets as being individual or shared and active or passive.

The compiler estimates that the model above contains an active/active synchronisation on the activity c where P performs the activity at rate r_3 and Q performs it at rate r_4. However, no such active/active synchronisation would ever take place because the local state P_2 is actually unreachable. This follows because Q would be required to first synchronise on a b activity with P but Q can never perform a b activity at all, only c. Faulting models such as this which appear to have an active/active synchronisation which they actually do not are not very problematic because this problem always indicates the presence of unreachable states (dead code) in the model, probably signalling an error on the part of the modeller.

We now present a case study which demonstrates the use of these tools.

5. Case study: hierarchical cellular network

The hierarchical cellular network consists of two tiers of cells, a level of macrocells overlying a level of microcells. In this study, we consider the Manhattan model [14,15] where the reuse pattern is based on a five squared microcell cluster, a central cell surrounded by four peripheral cell (Fig. 5). This model takes its name from the city of

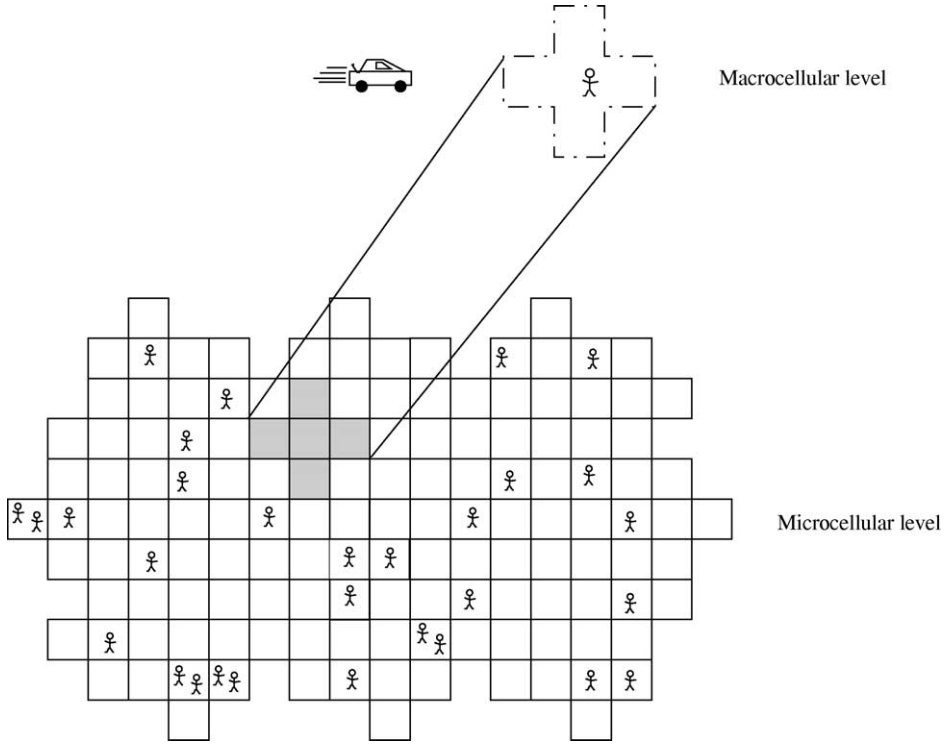

Fig. 5. Reuse pattern of Manhattan model.

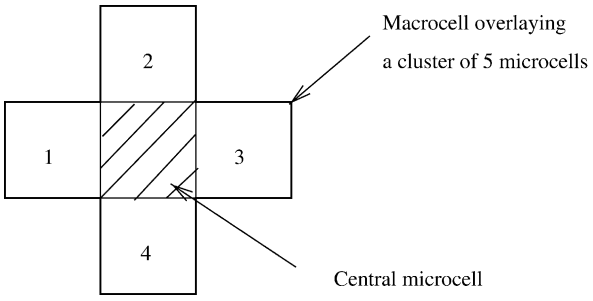

Fig. 6. The cluster model.

Manhattan, which consists of square-blocks, representing buildings, with streets in between them.

As in this model each microcell has four neighbouring cells, we consider a microcell cluster model composed of a central microcell surrounded by four peripheral cells as shown in Fig. 6. We consider a FCA scheme (Fixed Channel Allocation [16]), where S channels are distributed among the different cells. Let c_j, $j=1,\ldots,5$ be the capacity of microcell j and c_0 the capacity of the macrocell.

Considering a homogeneous system in statistical equilibrium, any cluster of microcells overlaid by a macrocell has statistically the same behaviour as any other cluster of microcells overlaid by a macrocell. We use this observation to decouple a cluster from the rest of the system. That is, we can analyse the overall system by focusing on a given cluster under the condition that the neighbouring clusters exhibit their typical random behaviour independently.

We consider two types of customers inside the cluster, the new calls and the handover requests (ongoing calls). External arrivals to the cluster consist of the handover requests coming from other clusters and the new calls initiated in that cluster. We assume that the handover requests coming from other clusters may occur only in the macrocell or the peripheral microcells. We consider that these arrivals may never occur in the central microcell.

In this study, new calls can be assigned only to the microcell level. Moreover, we consider a hierarchical cellular network using an overflow strategy but without reversible capability, except for the external arrivals to the macrocell. A request, either a new call or a handover request, initiated at the microcell level is served in its originating microcell if a channel is available. Otherwise, according to the overflow strategy, the request is overflowed to the upper level and is satisfied if a channel is free at this level. In the case where all channels are busy at both levels, the request is dropped (in the case of a handover) or blocked (in the case of a new call).

This system is studied under the usual Markovian assumptions. New call and handover request arrivals follow a Poisson process. We assume that the average new call arrival rates and the handover arrival rates are the same for all cells in the network. The session duration which represents the duration of a communication is modelled by a service time which is exponentially distributed with parameter μ. The amount of time that a user remains within a coverage cell of a given base station, called *dwell-time*, is assumed to be exponentially distributed with parameter α.

In Section 5.1, we present the UML model corresponding to this system.

5.1. The UML model

In the model, the external arrival process is represented by the event *in* by the cells. The arrival rate is assumed to be λ_1 in the macrocell, λ_2 in the peripheral microcells and λ_3 in the central microcell.

Because of the different types of cells (macrocell, peripheral microcell, central microcell) and the topology of the network, we make a distinction between handover requests generated by the cluster itself (Fig. 7). This distinction is based on the cell type this request originated from and the cell type satisfying this request, which means the cell where the ongoing call has to be transferred to. Thus, the arrival process of these customers is represented by the event *handoff* indexed by the type of handover request as follows:

- $handoff_{down}$ represents the transfer of a call from the macrocell to a microcell. This call is a handover request coming from outside the cluster to the macrocell and because all its channels are busy, it has to be transferred to a microcell,
- $handoff_{up}$ represents the transfer of a call from a microcell to the macrocell. This call may be either a new call or a handover request coming from outside the cluster to the microcell and because all channels in the microcell are busy, it has to be transferred to the

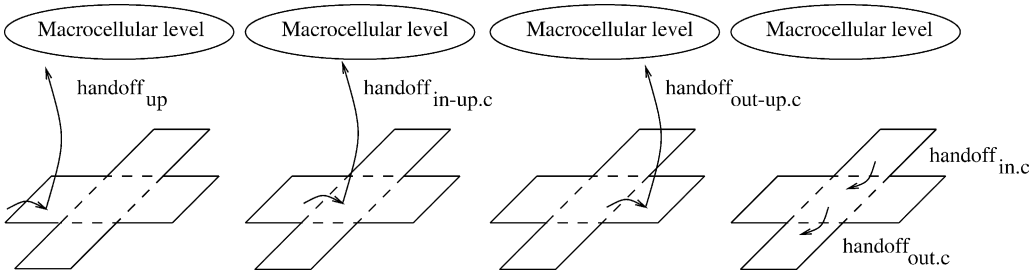

Fig. 7. The handoff requests graph.

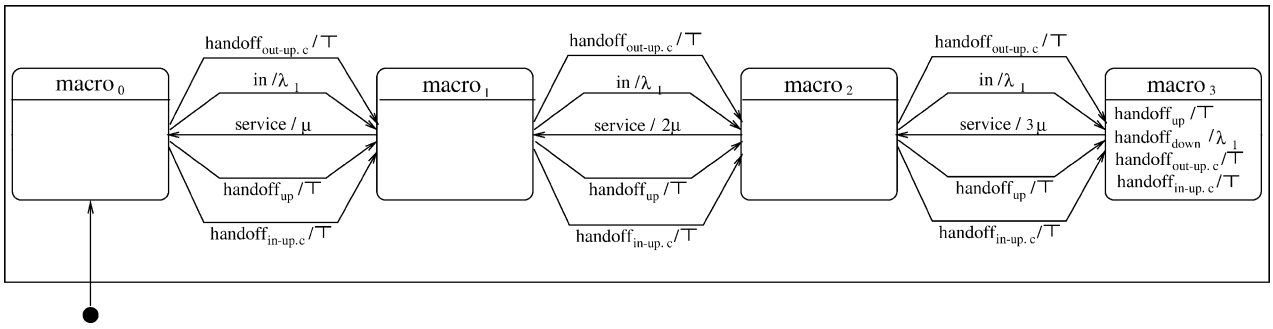

Fig. 8. The state diagram of *macro*.

macrocell. The rate associated with this event is the external arrival rate to the microcell,

- *handoff*$_{in.c}$ is the event which triggers the transfer of an ongoing call from one of the peripheral microcells to the central microcell,
- in contrast, *handoff*$_{out.c}$ represents the transfer of an ongoing call from the central microcell to one of the four peripheral microcells,
- *handoff*$_{in\text{-}up.c}$ represents the case where an ongoing call coming from a peripheral microcell and entering the central microcell, is then transferred to the macrocell because all channels of the central microcell are busy,
- *handoff*$_{out\text{-}up.c}$ models the arrival of an ongoing call from the central microcell to a peripheral microcell and because all channels of this cell are busy, the handover call is overflowed to the macrocell.

As the process behind the four last *handoff* events is the same, the corresponding rate is also the same and it is denoted by α (representing the mean dwell-time in a microcell). In all cells, the service process is represented by event *service*. As the service rate in each cell is assumed to be μ, when there are i, $1 \leq i \leq c_k$, customers in a cell, the event *service* is of rate $i\mu$.

In the following, we present the state diagrams of the different components of our network and the collaboration diagram showing the interactions between these components.

5.1.1. The state diagrams

We denote by *macro* the macrocell overlying the cluster of microcells. *Microc* and *microj*, denote the central microcell and a peripheral microcell, respectively. For the sake of readability of the different state diagrams, we limit the total number of channels to $S=18$ and these channels are fairly shared by the different cells: $c_j=3$, $j=0,\ldots,5$.

The state diagram of *macro* is described in Fig. 8 where $macro_i$, $i=0,\ldots,3$, is the state where i channels are busy.

A transition from state $macro_i$ to state $macro_{i+1}$ denotes the arrival of either an ongoing call from other macrocells (*in*) or from one of the microcells with *handoff*$_{up}$, *handoff*$_{in\text{-}up.c}$ or *handoff*$_{out\text{-}up.c}$. A natural termination of a call is represented by a transition from state $macro_i$ to state $macro_{i-1}$ with *service*.

When all channels are busy (state $macro_3$), if a handover call arrives from the microcells, the call is dropped and thus lost. Similarly, if an external handover call arrives when all channels are busy, the call is blocked and lost.

The state diagram of *microj* is described in Fig. 9 where $microj_k$, $k=0,\ldots,3$, is the state where k channels are busy.

A transition from state $microj_k$ to state $microj_{k+1}$ denotes an external arrival call (new call or a handover request from other clusters) with action *in*, an ongoing call from the central microcell (*handoff*$_{out.c}$) or from the macrocell (*handoff*$_{down}$). The departure of a call from a peripheral microcell to the central microcell (*handoff*$_{in.c}$) or to the macrocell via the central (*handoff*$_{in\text{-}up.c}$) is depicted in the diagram by a transition from $microj_k$ to state $microj_{k-1}$. A similar transition with *service* models a natural termination of a call. As for the *macro*, all arrivals when all channels of a peripheral cell are full (state $microj_3$) are lost.

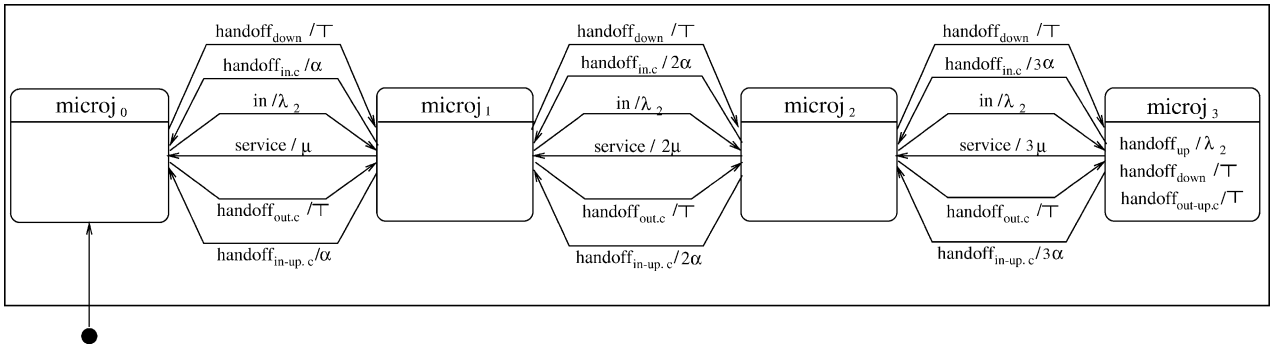

Fig. 9. The state diagram of a peripheral microcell *microj*.

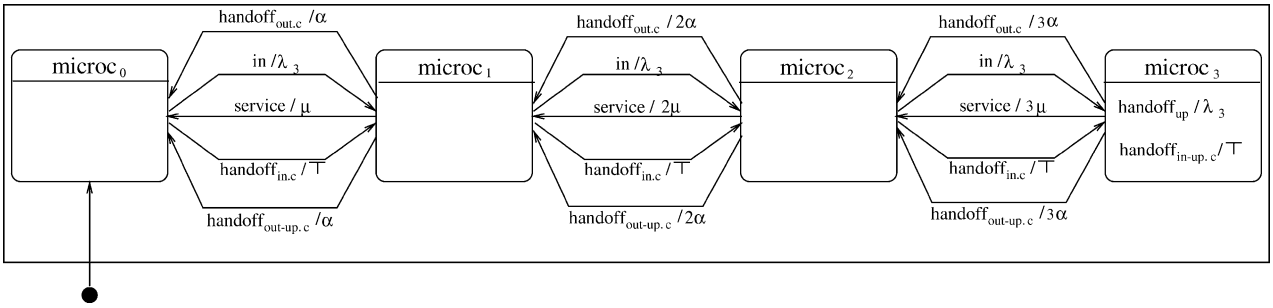

Fig. 10. The state diagram of the central microcell *microc*.

The state diagram of the central microcell *microc* is described in Fig. 10 where $microc_i$, $i=0,\ldots,3$, is the state where i channels are busy.

In this diagram, a transition from state $microc_i$ to state $microc_{i+1}$ denotes the arrival of either a new call (*in*) or an ongoing call from a peripheral microcell ($handoff_{in.c}$). A call leaving the central microcell for a peripheral microcell ($handoff_{out.c}$) or for the macrocell via a peripheral one ($handoff_{out\text{-}up.c}$) is depicted by a transition from $microc_i$ to state $microc_{i-1}$.

5.1.2. The collaboration diagram

In the network, the peripheral microcells will behave independently, but will synchronize with the central microcell when there are handoff requests from one to another. Similarly, both peripheral and central microcells have to synchronize. These interactions of the different components of the network are recorded in the collaboration diagram given in Fig. 11.

5.2. The PEPA model

The PEPA model extracted from the UML model consists of six components, *macro*, *microc* and *microj*, $1 \leq j \leq 4$. These components are defined in Fig. 12.

The system is formed by the co-operation of *macro* and the different microcells. Since the four peripheral microcells proceed independently, and co-operate with the central microcell, the system is defined as follows:

$$System \stackrel{def}{=} ((micro1_0 || \ldots || micro4_0) \underset{L}{\bowtie} microc_0) \underset{K}{\bowtie} macro_0$$

where L is the set of activities on which the central microcell and the peripheral microcells must synchronize. The set K contains the activities on which the macrocell and the microcells must synchronize. These two co-operation sets are defined as follows:

$$L = \{handoff_{in.c'}\ handoff_{out.c}\}$$

$$K = \{handoff_{up'}\ handoff_{in-up.c'}\ handoff_{out-up.c'}\ handoff_{down}\}$$

5.3. Processing the model

Solving this model requires using our extractor and reflector tools and the PRISM model checker. We solve the model for $c_j=8$, $j=0,\ldots,5$. This implies that there are 48 channels shared between 6 cells. The CTMC corresponding to this model has more than a quarter of a million states (actually 262,144 states) and the longest part of the process is generating and solving this CTMC with PRISM. We used PRISM v1.3 for this with its Hybrid solution engine and its Jacobi solver. The total storage for matrix and solution vectors built for the model required 6208 kb of memory and the solver found the solution after 466 iterations. This took 45.31 s on a 1.6 GHz Pentium IV with 256 Mb of memory. The fact that this runtime is so short means that these tools can be used by a software developer with no more significant impact on development time than that spent on the edit–compile–run cycle in software development. We think that this is an encouraging indicator for this method of software performance analysis.

A screenshot showing the reflected results in the UML model can be seen in Fig. 13. On each diagram state, we now have the name of the state and, between the brackets, a performance measure related to this state. In this example, we have the steady-state residence probability, expressed as a percentage, for each state.

5.4. Performance analysis and model-checking

For such a system, the performance measures of interest are the blocking probability of new calls and the dropping handover probability. The blocking probability is defined as the probability that a new call is denied access to a channel,

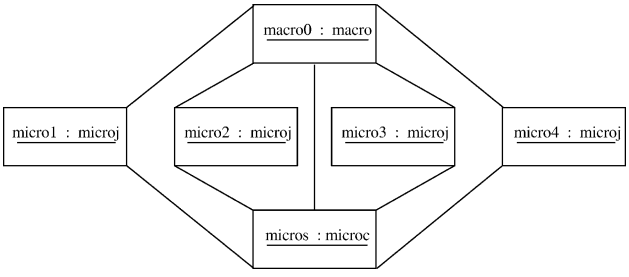

Fig. 11. The collaboration diagram.

Component *macro*:

$$
\begin{aligned}
macro_0 &\stackrel{def}{=} (in, \lambda_1).macro_1 + (handoff_{up}, \top).macro_1 + (handoff_{in\text{-}up.c}, \top).macro_1 \\
&\quad + (handoff_{out\text{-}up.c}, \top).macro_1 \\
&\vdots \\
macro_i &\stackrel{def}{=} (in, \lambda_1).macro_{i+1} + (service, i\mu).macro_{i-1} + (handoff_{up}, \top).macro_{i+1} \\
&\quad + (handoff_{in\text{-}up.c}, \top).macro_{i+1} + (handoff_{out\text{-}up.c}, \top).macro_{i+1} \\
&\vdots \\
macro_{c_0} &\stackrel{def}{=} (service, c_0\mu).macro_{(c_0-1)} + (handoff_{down}, \lambda_1).macro_{c_0} \\
&\quad + (handoff_{up}, \top).macro_{c_0} \\
&\quad + (handoff_{out\text{-}up.c}, \top).macro_{c_0} + (handoff_{in\text{-}up.c}, \top).macro_{c_0}
\end{aligned}
$$

Component *microj*: $1 \leq j \leq 4$

$$
\begin{aligned}
microj_0 &\stackrel{def}{=} (in, \lambda_2).microj_1 + (handoff_{out.c}, \top).microj_1 + (handoff_{down}, \top).microj_1 \\
&\vdots \\
microj_k &\stackrel{def}{=} (in, \lambda_2).microj_{(k+1)} + (handoff_{out.c}, \top).microj_{(k+1)} \\
&\quad + (service, k\mu).microj_{(k-1)} + (handoff_{in\text{-}up.c}, k\alpha).microj_{(k-1)} \\
&\quad + (handoff_{down}, \top).microj_{(k+1)} + (handoff_{in.c}, k\alpha).microj_{(k-1)} \\
&\vdots \\
microj_{c_j} &\stackrel{def}{=} (service, c_j\mu).microj_{(c_j-1)} + (handoff_{up}, \lambda_2).microj_{c_j} \\
&\quad + (handoff_{in\text{-}up.c}, c_j\alpha).microj_{(c_j-1)} + (handoff_{in.c}, c_j\alpha).microj_{(c_j-1)} \\
&\quad + (handoff_{down}, \top).microj_{c_j} + (handoff_{out\text{-}up.c}, \top).microj_{c_j}
\end{aligned}
$$

Component *microc* :

$$
\begin{aligned}
microc_0 &\stackrel{def}{=} (in, \lambda_3).microc_1 + (handoff_{in.c}, \top).microc_1 \\
&\vdots \\
microc_i &\stackrel{def}{=} (in, \lambda_3).microc_{(i+1)} + (service, i\mu).microc_{(i-1)} + (handoff_{in.c}, \top).microc_{(i+1)} \\
&\quad + (handoff_{out\text{-}up.c}, i\alpha).microc_{(i-1)} + (handoff_{out.c}, i\alpha).microc_{(i-1)} \\
&\vdots \\
microc_{c_5} &\stackrel{def}{=} (service, c_5\mu).microc_{(c_5-1)} + (handoff_{in\text{-}up.c}, \top).microc_{c_5} \\
&\quad + (handoff_{up}, \lambda_3).microc_{c_5} \\
&\quad + (handoff_{out.c}, c_5\alpha).microc_{(c_5-1)} + (handoff_{out\text{-}up.c}, c_5\alpha).microc_{(c_5-1)}
\end{aligned}
$$

Fig. 12. PEPA components extracted from the UML model.

because all channels accessible by new calls at both cellular levels are busy. This probability is computed by considering the states where the new calls are lost. A new call is lost if it arrives at a full microcell and the $handoff_{up}$ activity fails because the macrocell is also full.

The dropping handover probability is defined as the probability that a call in progress is blocked due to handoff failure during its communication. In other words, the ongoing call is blocked because there is no available channel in the cells it may be transferred to. This probability is computed by considering the states where the activities $handoff_{out\text{-}up.c}$ and $handoff_{in\text{-}up.c}$ may fail, states which correspond to a full macrocell.

The impact of different parameters of the system on these probabilities can be investigated. Among these parameters, we can find the channel allocation scheme used, the user mobility and the overflow strategy [15]. For example, Figs. 14 and 15 show the results we obtain when investigating the impact of the channel allocation schemes on the blocking and dropping probabilities. Two allocation schemes are considered, a symmetric scheme where each cell has 7 channels and a non-symmetric one where each microcell has 5 channels and macrocell has 17 channels.

In Fig. 14, we can see that, with the first channel assignment scheme, where all cells have the same capacity (7), the dropping handover probabilities obtained are higher than those obtained when we consider the second scheme. This is due to the fact that all channels of a macrocell may be requested by all microcells of the cluster. Indeed, the role of the macrocell is to provide additional capacity with a pool of overflow channels to be shared between all microcells. Thus, any microcell may request a transfer of an ongoing call to the macrocell and this request has more chance of being satisfied if an important channel number is assigned to the macrocell. In other words, even if we have less channels in each microcell (5 in the second scheme) which may increase the number of handovers, the availability of a greater number of channels (17) in the macrocell will allow

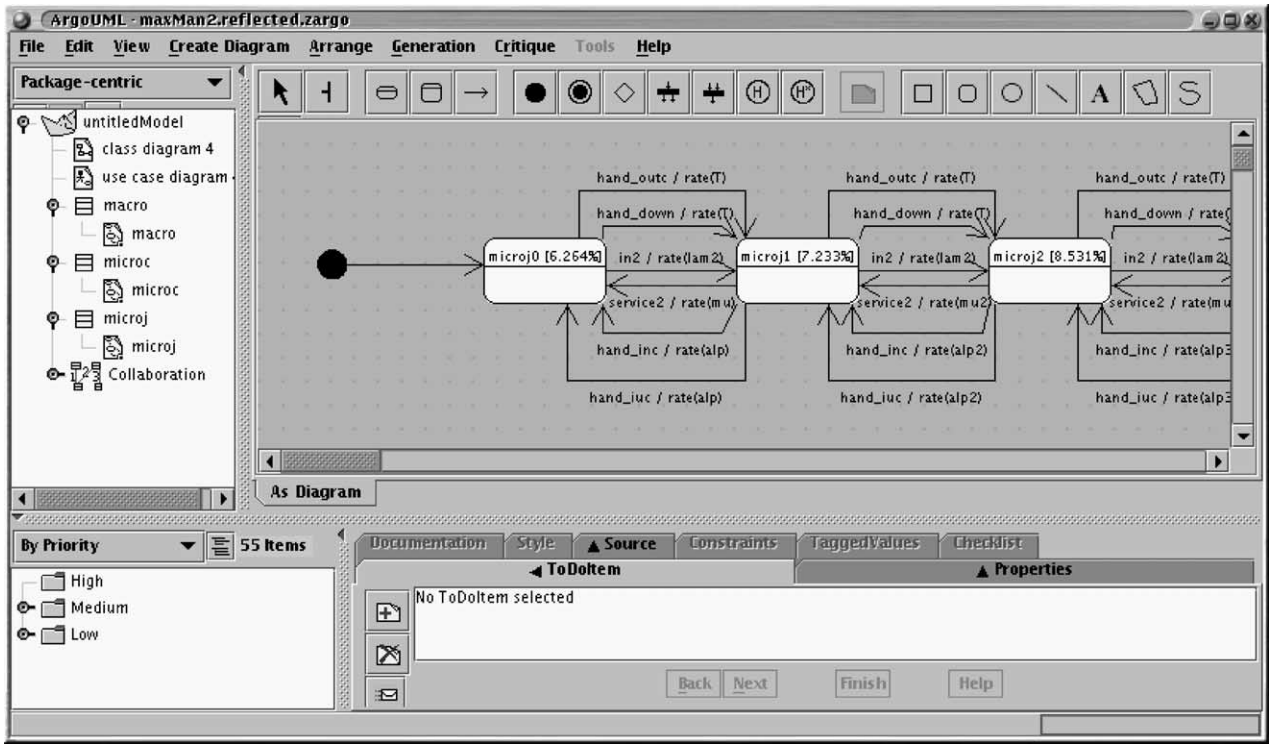

Fig. 13. The UML model with performance information added viewed in ArgoUML.

the macrocell to provide for a greater number of transfer requests of ongoing calls. The figure shows also that for low loads, the difference between the curves is important and that difference decreases considerably for heavy loads. The reason is that when the system is heavily loaded, all channels of the macrocell will be inclined to be occupied and it does not make a big difference to consider one channel assignment scheme rather than the other one.

In Fig. 15, we can observe the same phenomena as in the previous figure. New call blocking probabilities obtained when considering cells with the same capacity are greater than those obtained when considering the channel

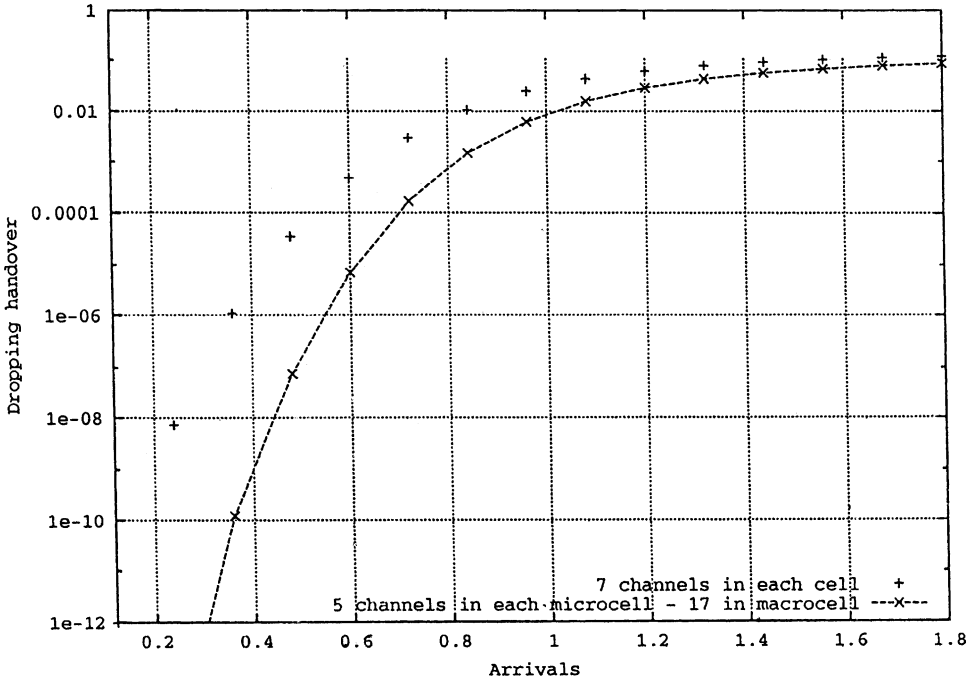

Fig. 14. Dropping handover probabilities.

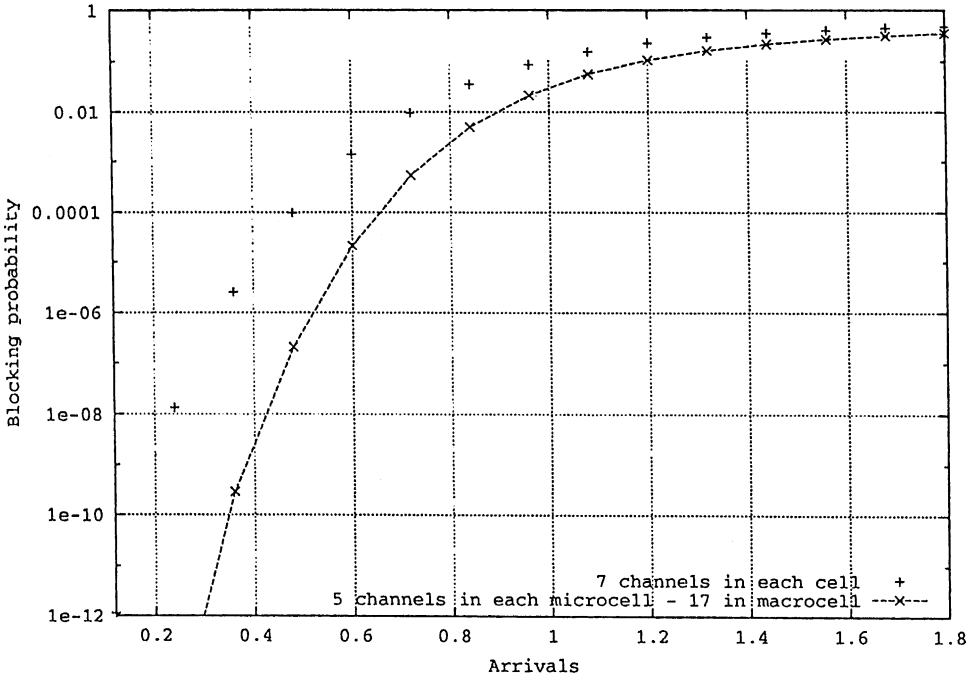

Fig. 15. New call blocking probabilities.

assignment scheme where the macrocell has an important channel number compared to microcells. This similarity is due to the fact that new calls may be assigned only to the microcellular level. That means that when all channels of the microcell in which this call has been generated are busy, the microcell engages in a $handoff_{up}$ activity. The new call is treated as an ongoing call.

Exploring symmetric versus asymmetric resource allocation schemes between both cellular levels shows that a channel allocation scheme assigning more resources to the macrocellular level yields better results.

CSL model-checking of CTMCs allows the user to check performability properties, which combine probabilities, behaviour and time. Using the PRISM tool, we model-checked a number of CSL formulae against our model of the Manhattan system. The most useful formula for performability analysis is a probabilistically quantified time-bounded until formula. We used formulae of the following generic form:

$$P_{\geq p}[pre\ \mathrm{U}_{\leq t}\ post]$$

The semantic meaning of such a formula is that with probability at least p, the system will be able to evolve from any state satisfying the *pre* formula to some state satisfying the *post* formula, in time at most t. We obtain the most information from such a formula by setting the probability as high as we can (as close to 1.0 as possible) and setting the time bound as low as we can (tighter time bounds are more informative than looser ones). We use these formulae to express *responsiveness* measures over the model, thereby learning how quickly the system can recover from error states, overloading, or other malfunctions.

PRISM's model-checking procedure for CSL formula determines the *number* of states in the reachable state space which satisfy the formula. This is strictly more informative than a simple Boolean-valued result indicating whether or not the formula is always satisfied.

The model analysis computes the size of the reachable state space as its first subcomputation. Given this result, we then may learn that from some determined proportion of this state space it is not possible for the system to respond within the stated time bound, with the required certainty. This might be all of the state space, none of it, or somewhere in between. In this way, model-checking CSL formulae and interpreting the results helps the modeller to learn more about the stochastic behaviour of the system and the progress of the probability mass through the state space over time.

We used PRISM's Hybrid engine and the CSL formulae were checked using the uniformisation and Fox-Glynn procedures from this engine. The results of our experiments are shown in Table 1. The first four columns give information about the formula which was checked (the p, *pre*, t, and *post* components of $P_{\geq p}[pre\ \mathrm{U}_{\leq t}\ post]$) and the last two columns give the number of satisfying states and the time taken for PRISM to check this formula (on a 1.6 GHz Pentium PC running Red Hat Linux 7.2).

Table 1 demonstrates several typical properties of CSL model-checking. Firstly, having a larger state space satisfying the *pre*-formula leads to a longer run-time. Secondly, higher probability of certainty leads to a longer

Table 1
CSL model-checking results

Probability p	Formula *pre*	Time t	Formula *post*	#states satisfying	Run time in s
0.5	**true**	1.0	*microcellFull*	262,144 (*all*)	6.453
0.5	**true**	0.001	*microcellFull*	229,376	4.708
0.995	**true**	0.001	*microcellFull*	229,376	5.255
0.995	*microcellFull*	0.001	$\neg$*microcellFull*	229,376	4.543
0.25	*allCellsFull*	0.1	$\neg$*allCellsFull*	262,144 (*all*)	0.947
0.5	*allCellsFull*	0.001	$\neg$*allCellsFull*	262,136	1.0
0.75	*allCellsFull*	1.0	$\neg$*allCellsFull*	262,144 (*all*)	0.843

run-time. Thirdly, tighter time bounds reduce the number of states satisfying a formula.

6. Related work

Work which is similar in spirit to our own approach is that of Petriu and Shen [17] where a layered queuing network model is automatically extracted from an input UML model with performance annotations in the format specified by a special-purpose UML profile [18]. We do not follow the same UML profile because it is not supported by our modelling tool. Additionally, the performance evaluation technology which we deploy (process algebras and BDD-based solution) is quite different from layered queueing networks.

Another performance engineering method which is similar to ours is that of Lòpez-Grao et al. [19] where UML diagrams are mapped into GSPNs which can be solved by GreatSPN. We use different UML diagrams types from these authors and, again, a different performance evaluation technology. Stochastic Petri nets and stochastic process algebras have different, but complementary, modelling strengths [20].

A different approach again is taken by Lindemann et al. [21] who map state and activity diagrams into generalised semi-Markov processes which can be solved by DSPNexpressNG. Their use of GSMPs provides an improvement in expressive power over our use of CTMCs, allowing deterministic delays in addition to exponentially distributed ones. However, this increased expressivity comes at the cost that GSMPs are less easily amenable to numerical solution than CTMCs.

One feature of our work, which is distinctive from both of the above, is the role of a *reflector* in the system to present the results of the performance evaluation back to the UML modeller in terms of their input model. We consider this to be a strength of our approach. We do not only compile a UML model into a performance model, we also present the results back to the modeller in the UML idiom.

7. Conclusions

We have described a component-based method of linking a collection of software tools to facilitate automated processing of UML performance models. The connectors in this method are the extractors and reflectors which we have developed. We have applied the tools to the analysis of a realistic model of a hierarchical cellular telephone network.

This approach to modelling allows the modeller to access a powerful and efficient solution technology without having to master the details of unfamiliar modelling languages such as process algebras and reactive modules. Our experience of using the PEPA and PRISM tools has been uniformly good.

One of the decisions which we have had to take in this work was the choice of UML diagrams and metaphors to employ. In part our choice in this was restricted by the degree of support offered by the UML modelling tool which we used (ArgoUML). However, the outcome of this was that it directed us to use familiar and well-understood parts of the UML modelling notation. One of our motivations for this work is reducing the potential for error in early stages of the performance modelling process and we consider that this outcome is supported by this influence to use the well-understood parts of UML.

It is not the case that an inexperienced modeller can use our system to compute any performance measure that they wish without needing any understanding of the abstraction, modelling and mathematical analysis at work in performance prediction and estimation. However, we hope that we have gone some way to providing automated support for computing simple performance measures and to circumventing an unnecessary notational hurdle if this was acting as an impediment to the understanding and uptake of modern performance analysis technology.

The dependability and safety of computer-based systems is a complex issue with many opposing and sometimes conflicting aspects. In this paper, we have focused on the quantitative aspects of system dependability known as *performability* because they lie on the border between performance and dependability. We have the view that it is sometimes the case that quantitative analysis takes second place to qualitative analysis of systems. However, safe, well-engineered systems need to deliver reliable services in a timely fashion with good availability. For this reason, we view quantitative analysis techniques as being as important as qualitative ones.

We have recently developed an extension of the PEPA stochastic process algebra where PEPA components are

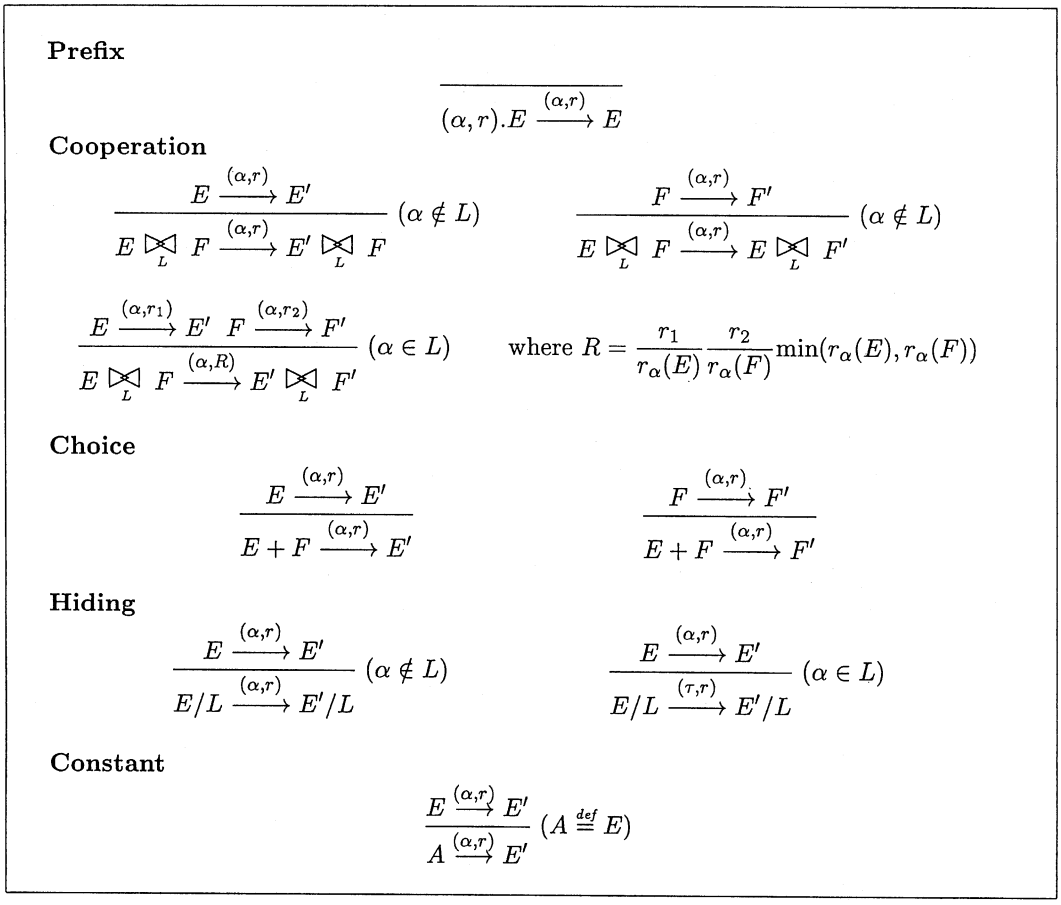

Fig. 16. The operational semantics of PEPA.

used as coloured tokens in a stochastic Petri net. The resulting formalism is called PEPA nets [23]. Our future work is to integrate the PEPA nets formalism with our extractor and reflector tools. Given an extended UML tool, which supports the forthcoming UML 2.0 design, we would be able to map the extended in UML 2.0 activity diagrams onto PEPA nets for analysis purposes. The activity diagrams in UML 2.0 are given a semantics which is based on Petri nets and queueing theory and are intended for analyses such as ours. An algorithm translating PEPA nets models into the PEPA formalism has already been developed and implemented [22]. Using this, it would be possible to take extended activity diagrams through to analysis by PRISM using the method followed in this paper.

Acknowledgements

The authors are supported by the DEGAS (Design Environments for Global ApplicationS) project IST-2001-32072 funded by the FET Proactive Initiative on Global Computing. The authors thank Gethin Norman and David Parker of the University of Birmingham for the implementation of the PEPA process algebra combinators in the PRISM model checker. Jane Hillston and David Parker provided helpful comments on an earlier draft of this paper. The anonymous referees of the SAFECOMP 2003 conference and of this special issue provided many helpful comments which have led us to improve this paper.

Appendix. Structured operational semantics for PEPA

The semantic rules, in the structured operational style, are presented in Fig. 16; the interested reader is referred to Ref. [6] for more details. The rules are read as follows: if the transition(s) above the inference line can be inferred, then we can infer the transition below the line. The notation $r_\alpha(E)$ which is used in the third co-operation rule denotes the apparent rate of α in E.

References

[1] Meyer JF. Closed-form solutions of performability. IEEE Trans Comput 1982;31-7:648–57.
[2] Tigris.org project. ArgoUML: a modelling tool for design using UML. Web page and documentation at http://argouml.tigris.org; 2002.
[3] Gilmore S, Hillston J. The PEPA Workbench: a tool to support a process algebra-based approach to performance modelling.

Proceedings of the seventh international conference on modelling techniques and tools for computer performance evaluation, LNCS 794, Vienna, May 1994 p. 353–68; http://www.dcs.ed.ac.uk/pepa.

[4] Kwiatkowska M, Norman G, Parker D. PRISM: probabilistic symbolic model checker. In: Field T, Harrison P, Bradley J, Harder U, editors. Proceedings of the 12th international conference on modelling techniques and tools for computer performance evaluation (TOOLS'02), London, April, LNCS 2324, 2002. p. 200–4.

[5] Object Management Group. Unified Modeling Language, v1.4. OMG document number: formal/2001-09-67, March; 2001.

[6] Hillston J. A compositional approach to performance modelling. Cambridge, MA: Cambridge University Press; 1996.

[7] Alur R, Henzinger TA. Reactive modules. Formal Meth Syst Des: Int J 1999;15-1:7–48. July.

[8] Canevet C, Gilmore S, Hillston J, Prowse M, Stevens P. Performance modeling with UML and stochastic process algebras. IEE Proc: Comput Dig Tech 2003;150(2):107–20. March.

[9] Parker D. PRISM 1.3 User's Guide, University of Birmingham, http://www.cs.bham.ac.uk/~dxp/prism, February; 2003.

[10] Somenzi F. CUDD: CU Decision Diagram Package, Department of Electrical and Computer Engineering, University of Colorado at Boulder, http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html, February; 2001.

[11] Aziz A, Sanwal K, Singhal V, Brayton R. Verifying continuous time Markov chains. Computer-aided verification, LNCS 1102. Berlin: Springer; 1996 p. 169–276.

[12] Milner R, Tofte M, Harper R, MacQueen D. The definition of standard ML: revised 1997. New York: The MIT Press; 1997.

[13] Benton N, Kennedy A. Interlanguage working without tears: blending SML with Java. Proceedings of the fourth ACM SIGPLAN international conference on functional programming, Paris, France, September 1999.

[14] Kulavaratharasah MD, Aghvami AH. Teletraffic performance evaluation of microcellular personal communication networks (PCN's) with prioritized handoff procedures. IEEE Trans Vehicul Technol 1999;48-1:137–52.

[15] Fourneau J-M, Kloul L, Valois F. Performance modelling of hierarchical cellular networks using PEPA. Perform Eval 2002;50: 83–99.

[16] Katzela I, Naghshineh M. Channel assignment schemes for cellular mobile telecommunication systems: a comprehensive survey. Proc IEEE 1994;82-9:1398–430.

[17] Petriu DC, Shen H. Applying the UML performance profile: graph grammar-based derivation of LQN models from UML specifications. In: Field AJ, Harrison PG, editors. Proceedings of the 12th international conference on modelling tools and techniques for computer and communications system performance evaluation, LNCS 2324, London, April, 2002. p. 159–77.

[18] Selic B, Moore A, Woodside M, Watson B, Bjorkander M, Gerhardt M, et al. Response to the OMG RFP for schedulability performance, and time. OMG document number: ad/2001-06-14, June; 2001.

[19] López-Grao JP, Merseguer J, Campos J. From UML activity diagrams to stochastic Petri nets: application to software performance analysis. Proceedings of the seventeenth international symposium on computer and information sciences. Orlando, FL: CRC Press; 2002 p. 405–9.

[20] Donatelli S, Hillston J, Ribaudo M. A comparison of performance evaluation process algebra and generalized stochastic Petri nets. Proceedings of the sixth international workshop on Petri nets and performance models, Durham, North Carolina 1995.

[21] Lindemann C, Thümmler A, Klemm A, Lohmann M, Waldhorst O. Performance analysis of time-enhanced UML diagrams based on stochastic processes. Proceedings of the third international workshop on software and performance (WOSP), Rome, Italy 2002 p. 25–34.

[22] Gilmore S, Hillston J, Kloul L, Ribaudo M. Software performance modelling using PEPA nets. ACM SIGSOFT fourth international workshop on software and performance (WOSP'04), Redwood City, California, January 2004.

[23] Gilmore S, Hillston J, Kloul L, Ribaudo M. PEPA nets: a structured performance modelling formalism. Perform Eval 2003;54-2:79–104. October.